LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-655453

Application-Agnostic Streaming Bayesian Inference via Apache Storm

T. Wasson, A. P. Sales

June 5, 2014

The 2014 International Conference on Advances in Big Data Analytics
Las Vegas, NV, United States
July 21, 2014 through July 24, 2014

Disclaimer

Application-Agnostic Streaming Bayesian Inference via Apache Storm

T. Wasson[1] **and A. P. Sales**[1]
[1]Data Analytics and Decision Sciences, Lawrence Livermore National Laboratory, Livermore, CA, USA

Abstract— *Given the increasing rates of data generation, along with increasingly ubiquitous sensors to measure it, analytical capabilities must be developed to keep pace. Although existing techniques have had success with some machine learning and statistical inferential tasks, these tasks are generally either limited in scope or are not capable of inferring the full probability densities (necessary for many sophisticated statistical analytical approaches) at a fast enough rate in order to match that of data arrival.*

We present a scalable framework for performing statistical density estimation on-the-fly on streams of data. This is achieve via a parallelized Apache Storm implementation of a particle learning algorithm. We demonstrate how to construct such an approach from the Storm primitives, and build upon these with novel contributions to Storm. Importantly, although our approach is exemplified via our particle learning framework, the ideas herein are generically applicable and agnostic of underlying modeling choices.

Source code available for Storm extensions upon request. Contact `wasson3@llnl.gov`.

Keywords: streaming data processing; density estimation; online statistical inference; Apache Storm; Bayesian statistics

1. Introduction

In virtually every domain of science, advances in data collection technologies have been greatly outpacing advances in the development of capabilities to satisfactorily store, process, and analyze such large volumes of data. There exists a consensus that there is a great deal to be learned and gained from these data, but how to make sense and extract meaningful information from these data streams of ever increasing volume and complexity remains a major challenge. While batch techniques for retrospective analysis, like Hadoop, Spark, and others, have resulted in powerful nuanced analytical approaches [1, 2], many data sources are too voluminous and arrive at too high a rate to be stored and post-processed via batch approaches and must be processed on-the-fly by streaming tools. Numerous techniques and software packages exist for performing a variety of machine learning and statistical inference tasks on streaming data [3, 4, 5, 6], but these are generally limited to relatively focused (albeit extremely useful) methods, such as classification via decision trees, support vector machines, or random forests, clustering, and regression techniques.

We present an implementation of a Bayesian density estimation technique via the stream processing framework Apache Storm, which can perform inference on-the-fly on data streams. Density estimation is performed via a sequential Monte Carlo algorithm, namely particle learning, and can be used in order to accomplish useful tasks such as clustering,classification, anomaly detection, and drift detection. The sequential nature of this algorithm allows the statistical model to be updated with each observations that arrives in the data stream, such that data need not be stored for batch processing. The Storm implementation allows for parallel computations within the streaming framework, substantially easing the computational burden entailed by these types of sophisticated statistical modeling. Finally, we also present novel extensions to Storm itself to facilitate this implementation. Altogether, we have developed a substantially powerful tool for performing sophisticated statistical inference on streaming data, allowing novel analyses and providing novel Storm additions.

The remainder of this article is structured as follows. We describe Storm and its utility in Section 2.1. In Section 2.2, we describe ParticleStorm, with the statistical model being described in Section 2.2.1, the capabilities and functionalities of the individual components of ParticleStorm topology being described in Section 2.2.2, and how these components work together to form a cohesive whole in Section 2.2.3. Our contributions to the Storm project are listed in Section 3, and our final remarks are given in Section 4.

2. Approach

2.1 Apache Storm

Apache Storm [7] is a distributed fault-tolerant real-time stream processing framework. Using Storm, arbitrary event-based functions can be calculated on-the-fly on inbound data streams, with the calculations spread across compute clusters, including easy deployment on cloud computing frameworks such as Amazon Web Services. Storm has been used effectively for a broad variety of applications [8], including various data analytics, machine learning tasks, and continuous computation tasks.

Storm includes a collection of fundamental software and terminology abstractions necessary to describe and implement its capabilities. A complete computational system in Storm is a directed graph, called a *topology*. The nodes of

the topology are *spouts*, which emit data, and *bolts*, which ingest and perform computations on data, optionally emitting results of these computations downstream to other bolts. Data and other communications within a topology are carried within *streams* of *tuples*. Streams may be subscribed to by any bolt, except for *direct streams*, in which the producer can decide explicitly which bolt is to receive a given tuple. Spouts and bolts may produce, and bolts may consume, any number of streams. Tuples are vectors of arbitrarily-typed elements, which may contain data or its derivations, or may contain messages important for the control schemes overlaid on topologies to enforce fault tolerance, exactly-once processing, and other necessary overhead important to a resilient processing framework.

Storm is written in Java and Clojure, and hence designed to run in Java Virtual Machines, but it supports a protocol for communication with external processes via its multi-language (*multilang*) protocol. Bolts that communicate with external processes for data processing via the multilang protocol are called *ShellBolts*, and will be discussed further in Section 2.2.2.

Recent development in Storm has been primarily focused on *Trident*, an abstraction allowing implementation of many commonly-desired use cases with dramatically less work necessary for the developer. However, Trident imposes restrictions to achieve these benefits, including removing the explicit definition of bolts and their stream subscriptions, and hence the ability to enforce cycles and build nuanced control schemas of the user's design. As such, Trident is eminently practical for the majority of tasks, but insufficient for some complex situations such as those discussed below.

Storm is one of many stream processing frameworks presently available, including Apache S4, Samza, Spark Streaming, and others. We pursued development with Storm because it allowed the finest-grained control of processing unit (bolt) heterogeneity, the greatest control over data streams (specifically, allowing arbitrary communications within the topology), and maturity and support in the community.

2.2 Storm for density estimation

Density estimation is a particularly powerful technique for learning from unstructured data, and we discuss here how it is carried out via particle learning, followed by a description of how particle learning can be implemented in Storm.

2.2.1 Streaming density estimation using particle learning

Consider a state-space model that is evolving over time, where the true underlying model state, x, is unobservable and information about it is only obtained via noisy measurements, y, at each time step, t. The state vector at time t, x_t, given all observed measurements up to that time step, $y_{1:t}$, can be estimated via its *filtering distribution*:

$$p(x_t|y_{1:t}) = \frac{p(y_t|x_t)p(x_t|y_{1:t-1})}{\int p(y_t|x_t)p(x_t|y_{1:t-1})dx_t}, \tag{1}$$

where the *predictive distribution* of state x_t given the observed measurements up to the previous time step, $y_{1:t-1}$, is given by

$$p(x_t|y_{1:t-1}) = \int p(x_t|x_{t-1})p(x_{t-1}|y_{1:t-1})dx_{t-1}. \tag{2}$$

In effect, Equation (1) is a simple application of Bayes theorem, where the predictive distribution $p(x_t|y_{1:t-1})$ is treated as the prior for x_t before the arrival of measurement y_t. In most cases, Equations (1) and (2) are analytically intractable, but can be approximated by particle filtering.

Particle filtering [9] is a sequential Monte Carlo method in which the current state variable is estimated by a weighted average of a set of random i.i.d. samples, called *particles*, from the state variables obtained by its filtering density given in Equation (1)[1]. As the number of particles increases, the particle filter approximation converges to the actual distribution.

Let $\{x_t^{(i)}\}_{i=1}^N$ be a set of N particles generated from the filtering distribution. The predictive distribution can be approximated by

$$p(x_t|y_{1:t-1}) \simeq \frac{1}{N}\sum_{i=1}^{N}\delta_{x_t^{(i)}}, \tag{3}$$

where $\delta_{x_t^{(i)}}$ is the Dirac delta function centered at $x_t^{(i)}$. By substituting (3) into (1), the filtering distribution can be approximated via a discretization of $p(x_t|y_{1:t})$ into particles $\{x_t^{(i)}\}_{i=1}^N$ with probabilities $\{w_t^{(i)}\}_{i=1}^N$,

$$\begin{aligned} p(x_t|y_{1:t}) &\simeq \sum_{i=1}^{N} w_t^{(i)}\delta_{x_t^{(i)}}, \\ w_t^{(i)} &= \frac{p(y_t|x_t^{(i)})}{\sum_{i=1}^N p(y_t|x_t^{(i)})} \end{aligned} \tag{4}$$

Particle filters operate simply by iterating between (3) and (4) at each time step with the arrival of new observations. A common shortcoming of particle filters is that the weights of particles in regions of high posterior density steadily increase to the point that eventually a single particle dominates the filter, and the weights of all other particles become negligible [10]. This so-called degeneracy problem can be directly quantified via the effective sample size of the particle set,

$$N_{\text{eff}} = \frac{N}{1 + \text{Var}(\{w_t^{(i)}\}_{i=1}^N)}, \tag{5}$$

such that the smaller the effective sample size of a filter, the more severe the degeneracy problem. A brute force solution

[1]To be precise, particle filtering algorithms entail numerical approximations of the joint posterior distribution $p(x_{1:n}|y_{1:n})$. Equation (1) shows only the marginal $p(x_n|y_{1:n})$ for simplicity.

to this problem is to use a very large N, but this leads to prohibitively large computational burdens. A more effective approach entails eliminating particles with small weight via a *resampling* step.

The resampling step stochastically enriches the particle set with high-importance particles, by eliminating particles with small importance weights. It works by sampling N particles with replacement from the set of particles $x_t^{(i)}$ according to their respective weights. The weights of the new generation of particles is set to $1/N$. Samples with large weights are likely to be drawn multiple times, whereas those with small weights are likely to be drawn very few times or not at all. Thus, particle filtering can be seen as a type of *survival of the fittest* algorithm, where higher weight particles are likely to produce more "offspring."

While resampling attenuates the degeneracy problem, it may lead to sample impoverishment. That is, because high-weight particles are likely to be drawn multiple times, over time the diversity of the samples is drastically reduced. Sample impoverishment is detrimental to the filter accuracy, as it results in worse approximation of the state distribution. Liu and Chen (1998) [11] provide a detailed discussion of the merits of resampling.

Particle learning (PL) [12] is a type of particle filtering algorithm that overcomes both the degeneracy and the sample impoverishment drawbacks of common particle filters. In fact, Carvalho *et al.* (2010) [12] demonstrated that PL's accuracy is not only superior to standard particle filter algorithms, but is comparable to MCMC samplers. PL improves upon traditional particle filtering algorithms in two ways:
Conditional sufficient statistics, s, are used to represent the posterior distribution of unknown parameters, θ, which is learned (hence, particle learning) as new observations arrive. The particles now are represented by $\{z_t^{(i)} = (x_t^{(i)}, s_t^{(i)}, \theta_t^{(i)})\}_{i=1}^N$, which are generated from the predictive distribution $p(x_t, \theta | y_{1:t-1})$ (likewise, $\widetilde{z}_t$ is sampled from the filtering distribution $p(x_t, \theta | y_{1:t})$.
The Resample-Propagate algorithm (shown in Table 1) is used in order to obtain exact samples from the particle approximation. Performing resample first and propagate second reduces approximation errors, because states are only propagated after being informed by the new observation t_{t+1}. Hence, only "good" particles are propagated.

We use the PL algorithm for composite mixture models, where each mixture component is a composite of independent distributions for each element of the response and predictor arrays. This approach enables modeling of data that includes multiple disparate feature types into a single probability model without resorting to complicated embeddings that would preclude sequential analysis. This model is described in detail in Sales *et al.* (2013) [13].

Table 1: Particle learning algorithm. Initialization is performed once, and steps 1 through 3 are repeated for each observation that arrives in the data stream.

Step	Task	Description
Step 0	Initialization	Set the starting values of the N particles $\{z_t^{(i)} = (x_t^{(i)}, \theta_t^{(i)})\}_{i=1}^N$
Step 1	Evaluation	Evaluate new observation y_{t+1} under the current model, $p(y_{t+1} \mid z_t^{(i)})$
Step 2	Resample	Resample $\{\widetilde{z}_t^{(i)}\}_{i=1}^N$ with weight $w_t \propto p(y_{t+1} \mid z_t^{(i)})$
Step 3	Propagate	$z_t^{(i)}$ from $p(z_t^{(i)} \mid \widetilde{z}_t^{(i)})$

2.2.2 Storm implementation

We have implemented streaming density estimation via particle learning in Storm to yield a tool entitled ParticleStorm. ParticleStorm functions in either inference mode, in which model parameters are updated after each data point is processed and resampling may or may not occur (though propagation always does), or evaluation mode, in which all parameters are fixed and data points are evaluated as quickly as possible under the current ensemble of models. Because particle learning in inference mode requires model parameter updates, and those updates require knowledge gained from the entire ensemble en masse, direct asynchronous communication between some (but not all) bolts is a necessity. Indeed, ParticleStorm has a number of characteristics that make it somewhat different from the majority of Storm frameworks, and in aggregate preclude the use of Trident. Specifically:

- ParticleStorm relies on an external C++ executable implementation of particle learning, entitled PF, modified to perform task-specific functions within the larger distributed Storm topology.
- ParticleStorm requires exactly-once computation of data points.
- ParticleStorm requires that between data points, parameters of some bolts are updated and possibly retrieved or overwritten.
- ParticleStorm must be switchable between inference and evaluation mode on-the-fly via an external control mechanism

Hence, ParticleStorm is constructed from the base spout and bolt abstractions.

It is important to note that Storm is quite complex, but much of that complexity is usually hidden from the developer, particularly for common or straightforward applications. For the implementation of ParticleStorm, we extended Storm to accommodate desirable properties, such as synchronicity and exactly-once data processing. Our extensions make heavy use of Storm's CoordinatedBolt class and other aspects of transactional topologies [14]. These include a number of control mechanisms, in terms of additional bolts and streams, to facilitate guaranteed exactly-once message

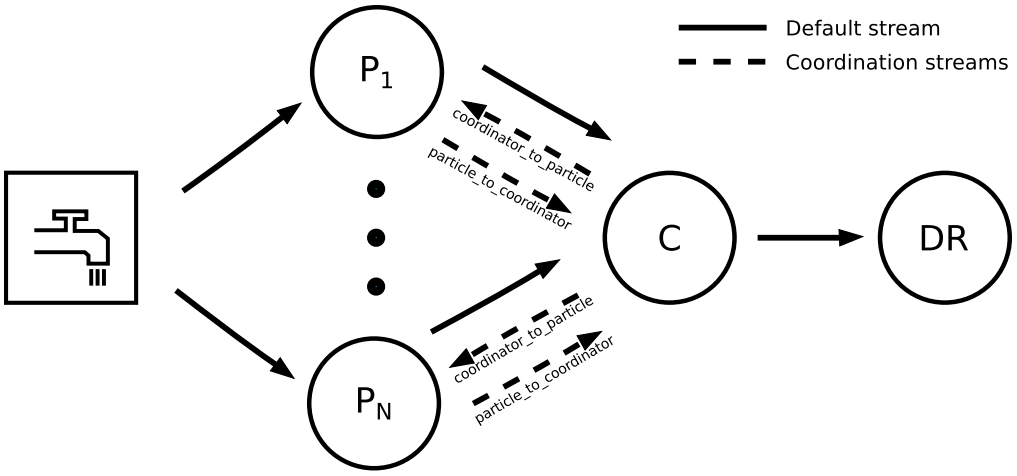

Fig. 1: ParticleStorm Storm topology. ParticleStorm is composed of a spout, a model-specific number of N ParticleBolts with one ParticleBolt per particle, a CoordinatorBolt, and a DataRecorder. The `default` stream flows through the entirety of the topology, carrying first data points and then log likelihoods derived from those data points. Two coordination streams connect each ParticleBolt to the CoordinatorBolt. Tuples of data are generated by the spout and passed to all ParticleBolts, where they are evaluated under each particle. The log likelihoods of the data points are passed on `default` to the CoordinatorBolt, which combines them and in turn passes the complete model log likelihood along to the DataRecorder, also on `default`. The coordination streams `coordinator_to_particle` and `particle_to_coordinator` are used to allow the CoordinatorBolt to communicate selectively with each ParticleBolt, performing operations including sending and retrieving model parameters during inference. Walkthroughs of typical processing runs are given in more detail in Section 2.2.3.

passing. For example, usage of CoordinatedBolts implies the creation of a wrapper CoordinatedBolt and its "delegate" bolt, being the bolt of the developer's design. The CoordinatedBolt intercepts communications to and from the delegate and, along with its own use of coordination tuples passed on implicitly-created coordination streams, tracks how many inbound tuples a delegate can expect to receive from its upstream bolts before it can be confident in having received all tuples exactly once.

Additionally, ParticleStorm relies on transactional spouts, necessary to ensure that an entire 'batch' of tuples is processed exactly once and to allow potential replaying of tuples upon failure, which transparently create separate spout coordinator and spout emitter tasks, and also use special-purpose batch initialization and commit tuples. Storm topologies are transparently augmented with an 'acker' bolt to handle acknowledgments of tuples and hence be able to trigger proper replaying of a data tuple if a descendant tuple fails. We explicitly acknowledge the importance and relevance of these usually-hidden complexities, but going forward, we choose to set them aside and focus on the explicit portions of our topology in subsequent descriptions and figures.

The (explicit) ParticleStorm topology (Figure 1) is composed of a spout, one or more ParticleBolts, a CoordinatorBolt, and a DataRecorder bolt, interconnected by several communications streams and one data stream. We describe the components of the topology here, and give examples of its functionality in Section 2.2.3 to illustrate how the components operate in concert to perform inference and evaluation.

Streams: ParticleStorm has three types of streams: `default`, `coordinator_to_particle`, and `particle_to_coordinator`, which function as follows.

- `default` carries data observations, represented as pipe-delimited feature vectors, and calculations derived from the data observations.
- `coordinator_to_particle` is a direct stream by which the CoordinatorBolt may communicate with ParticleBolts. This stream carries issued commands from the CoordinatorBolt, with replies expected to arrive on `particle_to_coordinator`. Tuples on this stream have three fields, and are of the form `[ID, command, content]`. `ID` is a unique identifier used to track responses to this command. `command` is one of `requestParameters`, `assignParameters` or `propagate`. When `command` is `requestParameters` or `propagate`, `content` is empty. When `command` is `assignParameters`, `content` contains the model parameters to be assigned to the destination ParticleBolt.
- `particle_to_coordinator` is a direct stream by which ParticleBolts may communicate with the CoordinatorBolt. This stream carries replies from ParticleBolts to the CoordinatorBolt, in response to commands issued on `coordinator_to_particle`. Tuples on this stream have two fields, and are of the form `[ID, content]`. As with `coordinator_to_particle`, `ID` is a unique identifier, and is identical to the `ID` in the `command` to which this tuple is responding. `content` is the response to the specific `command` received. When `command` is `assignParameters` or `propagate`, `content` is `ack`. When `command` is `requestParameters`, `content` contains the model parameters of this ParticleBolt.

ParticleStorm spout: The spout in ParticleStorm can be any TransactionalSpout [14], but must also incorporate the Storm Signals framework [15]. Storm Signals allow asynchronous communication with the spout outside of the traditional streams / tuples mechanism. This communication allows the spout to be paused and resumed as necessary, which is important during initialization and inference (described in Section 2.2.3), and allows a user to selectively pause the processing in the topology.

ParticleBolts: ParticleBolts are the fundamental source of parallelism employed in ParticleStorm. Each ParticleBolts hosts one particle in the particle learning model, and hence the number of ParticleBolts, equal to the number of particles, is model-dependent and determined at runtime. ParticleBolts extend ShellBolts, as the underlying modeling is done in the external PF binary executable. ParticleBolts subscribes to the `default` stream from the spout and `coordinator_to_particle` stream from the CoordinatorBolt, and outputs `default` and `particle_to_coordinator` streams. Data tuples received on `default` are scored under the modeled particle, and log likelihoods are subsequently emitted on `default`. Commands from the CoordinatorBolt are received on `coordinator_to_particle` and responses to those commands are emitted on `particle_to_coordinator`.

CoordinatorBolt: The CoordinatorBolt is the driver of ParticleStorm. There is exactly one CoordinatorBolt in the ParticleStorm topology, and it is responsible for tasking ParticleBolts to perform operations along with interpreting their output. Like ParticleBolts, the CoordinatorBolt extends ShellBolt and delegates work to PF. The CoordinatorBolt subscribes to the `default` and `particle_to_coordinator` streams from each ParticleBolt, and outputs `default` and `coordinator_to_particle` streams. In both evaluation and inference mode, the CoordinatorBolt receives log likelihoods from all ParticleBolts, weighing them appropriately to produce an overall model log likelihood for a data point, and emits that output. In inference mode, the CoordinatorBolt will determine whether a resample step is necessary. If so, the new vector of particles is sampled, and lists are created of particles to be overwritten and particles to provide parameters to do the overwriting. The CoordinatorBolt will emit a `requestParameters` command to each 'overwriting' particle, received the particle's parameters on `particle_to_coordinator`, and then emit an `assignParameters` command to overwrite the appropriate particle and await acknowledgment (indicating success), at which point the resample step is complete. ParticleBolts will then be tasked to update their parameters via a `propagate` command.

DataRecorder: The DataRecorder is responsible for processing, and potentially saving, the output of the data evaluation. It subscribes to the `default` stream from the CoordinatorBolt, which provides the log likelihood of each data point evaluated under the entire model. Our DataRecorder bolt can be set to either save results in an HDFS store or discard them and instead save the trained models.

2.2.3 ParticleStorm runtime modes and descriptions

As discussed previous, ParticleStorm operates in two discrete modes, being evaluation and inference. The two modes are largely identical in practice, with the primary differences being that in inference mode, resampling and propagation steps are included after each data point is evaluated.

As we describe the functionality of ParticleStorm, we make two simplifying choices for clarification: first, common underlying tuple chatter present in all Storm transactional topologies is omitted, and second, we present the steps in the algorithm as if they were sequential. Importantly, Storm is quite asynchronous, and handling this asynchrony in a way that is fault-tolerant and dependable is nontrivial. Indeed, correctly handling asynchrony was the motivation for much of our novel additions to Storm itself, discussed later in Section 3.

We describe evaluation and inference modes together in Table 5. First, all components take part in the Initialize process, described in Table 4. This process is composed of the CoordinatorBolt initializing the particle learning model, either from a given initial model or *de novo*, and transmitting the appropriate model parameters to each ParticleBolt via the AssignParameters process (Table 2). AssignParameters and RequestParameters (Table 3) are

Table 2: **AssignParameters** in ParticleStorm. Parameter assigning scheme. This occurs in initialization and inference mode.

Component	Action
CoordinatorBolt	**Emit direct** parameters on stream **`coordinator_to_particle`** to **ParticleBolt**
CoordinatorBolt	Wait for `ack`
ParticleBolt	**Receive** parameters on stream **`coordinator_to_particle`**
ParticleBolt	Set parameters
ParticleBolt	**Emit direct** `ack` on stream **`particle_to_coordinator`** to **CoordinatorBolt**
CoordinatorBolt	**Receive** `ack` on stream **`particle_to_coordinator`** from **ParticleBolt**
CoordinatorBolt	End wait

Table 3: **RequestParameters** in ParticleStorm. Parameter requesting scheme. This occurs in inference mode and optionally when saving models.

Component	Action
CoordinatorBolt	**Emit direct** `requestParameters` on stream **`coordinator_to_particle`** to **each ParticleBolt**
CoordinatorBolt	Wait for `parameters`
ParticleBolts	**Receive** `requestParameters` on stream **`coordinator_to_particle`**
ParticleBolts	**Emit direct** `parameters` on stream **`particle_to_coordinator`** to **CoordinatorBolt**
CoordinatorBolt	**Receive** `parameters` on stream **`particle_to_coordinator`** from **ParticleBolt**
CoordinatorBolt	End wait

Table 4: **Initialize** in ParticleStorm. This occurs at startup in both inference and evaluation modes.

Component	Action
CoordinatorBolt	Process initial model files or set parameters *de novo*
CoordinatorBolt	**AssignParameters** to all ParticleBolts
CoordinatorBolt	Enable **spout** via **Storm signals**

Table 5: Evaluation and inference mode functionality of ParticleStorm. We omit error checking and housekeeping tuples integral to all Storm topologies for clarity.

Control	Component	Action
	All components	**Initialize**
	Spout	**Receive enable** on stream **Storm signals**
For each data tuple		
	ParticleBolts	**Receive data** on stream **`default`**
	ParticleBolts	**Emit log likelihood** on stream **`default`**
	CoordinatorBolt	**Receive all log likelihoods** on stream **`default`**
	CoordinatorBolt	Calculate aggregate **log likelihood**
	CoordinatorBolt	**Emit aggregate log likelihood** on stream **`default`**
If inference mode		
If resample necessary		
	CoordinatorBolt	Calculate **Overwriting Particles**
	CoordinatorBolt	Calculate **Overwritten Particles**
	CoordinatorBolt	**RequestParameters** from **Overwriting Particles**
	CoordinatorBolt	**AssignParameters** to **Overwritten Particles**
End if		
	CoordinatorBolt	**Emit direct** `propagate` on stream **`coordinator_to_particle`** to **ParticleBolts**
End if		
	DataRecorder	**Receive aggregate log likelihood** on stream **`default`**
	DataRecorder	Record **aggregate log likelihood**
End for		

complementary functions by which the CoordinatorBolt sets or retrieves model parameters from the individual ParticleBolts. They involve communicating over the `coordinator_to_particle` and `particle_to_coordinator` direct streams to issue commands and receive responses.

After successful initialization, the CoordinatorBolt signals the spout via Storm Signals to indicate that the topology has been fully configured and is ready to process. At this point, the spout emits data tuples (one-at-a-time in ParticleStorm, as particle learning requires model updates per-data-point), which are received by all of the ParticleBolts via an 'all grouping' [16]. Each ParticleBolts evaluates the data tuple under its particle alone and emits the log likelihood on the `default` stream. The CoordinatorBolt receives the log likelihoods, aggregates them, and emits the overall log likelihood of the data point on the `default` stream, which the DataRecorder receives and records per the developer's design.

In evaluation mode, this data point is now fully processed, and the next data point is begun. In inference mode, however, the resample and propagate steps are undertaken first. If the CoordinatorBolt determines that resampling is necessary, a new vector of particles is calculated and differences from the previous vector are determined. Particles to spawn descendants, or overwriting particles, are queried for their parameters via RequestParameters. Particles to be overwritten then have these new parameters assigned to them via AssignParameters. Finally, regardless of whether resampling was necessary, the CoordinatorBolt emits a `propagate` command to each ParticleBolt on the `coordinator_to_particle` stream and awaits `ack` responses on the `particle_to_coordinator` stream. This marks the complete processing of this data point in inference mode, and the topology is now ready for the next data point.

3. Extensions to the Storm framework

In the process of developing ParticleStorm, it was necessary to develop a collection of additional capabilities for the Storm framework itself. These primarily concerned extension of preexisting capabilities to function inside of transactional topologies. Independent of the direct benefits of ParticleStorm itself, these extensions add useful functionality of Storm and will be contributed to the larger Storm community.

Although Storm allows for transactional topologies, and allows for ShellBolts, it does not allow for transactional ShellBolts. Hence, from the Storm ShellBolt class, we created a BatchShellBolt. This implied the modification of CoordinatedBolt to work around synchronization issues, because coordination tuples intrinsic to the functionality of CoordinatedBolt can otherwise be inadvertently processed before a data tuple is completely processed by the delegate shell process. Additionally, we extended the Storm multilang protocol to include 'housekeeping' commands between CoordinatedBolt and its delegate process, to inform the delegate of batch completion and allow the delegate to inform CoordinatedBolt of acknowledgment and completion of batch-finishing steps.

We also modified the ShellBolt class to allow direct execution of binary executables included in the Storm deployable uberjar. Previously, only system-level executables could be called, and they would be called on scripts included in the uberjar (e.g., Python or Ruby programs).

Finally, Storm Signals provides functionality to traditional Storm spouts, but not to transactional spouts, so we developed a transactional Storm Signals spout.

4. Discussion

As various aspects of our world are becoming increasingly measured with innumerable sensors of varying types, streaming data is becoming ubiquitous, and is vastly increasing in volume. Performing sophisticated analytics on these types of data is challenging and often infeasible because these computations are usually burdensome, and cannot keep up with the inflow rate of data. Storm, and other streaming frameworks, enable parallel processing of streaming data, allowing real-time analysis on data streams.

Here, we have presented ParticleStorm, an implementation of the particle learning algorithm in the Storm stream processing framework, a first of its kind. Particle learning is a sequential Monte Carlo algorithm that enables Bayesian statistical modeling to learn posterior probability densities via online inference at scale. In particular, ParticleStorm implements particle learning for composite mixture models, which allow nuanced models to be learned of phenomena with different numbers and different types of features, and to use those models to evaluate new data, with direct extension to tasks including clustering, classification, regression, anomaly detection, and drift detection.

Storm is still in its infancy, and although it has garnered a significant following, with users of the likes of Google, Yahoo, and Groupon, the literature on Storm is still scarce, which can present a high barrier to entry, especially for tasks outside of the traditional comfort zone addressed by Trident. One of the goals of this manuscript has been to reduce the entrance difficulty by providing a guide to other developers interested in uses of Storm beyond its most common utilities.

This implementation was possible via our extensions to the Storm project. In particular we have added the capability of Storm to delegate processing to external binary executables while still enforcing fault-tolerant exactly-once (transactional) processing. ParticleStorm is intentionally modular and easily extensible, allowing various pre- and post-processing extensions to be easily integrated. Its ideas, and the Storm extensions developed to allow its implementation, provide benefit for future efforts to use Storm, or other streaming frameworks, for complex statistical inference and machine learning tasks.

5. Acknowledgments

We would like to thank Vera Bulaevskaya and Daniel Merl for their invaluable contributions to this work. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] Apache Mahout: Scalable machine learning and data mining. https://mahout.apache.org/ [Accessed: 2014-04-19]

[2] Apache Spark - Lightning-Fast Cluster Computing. http://spark.apache.org/ [Accessed: 2014-04-19]

[3] Jubatus: Distributed Online Machine Learning Framework. http://jubat.us/ [Accessed: 2014-04-19]

[4] SAMOA. https://github.com/yahoo/samoa [Accessed: 2014-04-11]

[5] Storm-Pattern. https://github.com/quintona/storm-pattern [Accessed: 2014-04-11]

[6] Trident-ML. https://github.com/pmerienne/trident-ml [Accessed: 2014-04-11]

[7] Storm. http://storm.incubator.apache.org/ [Accessed: 2014-04-17]

[8] Storm applications in industry. https://github.com/nathanmarz/storm/wiki/Powered-By [Accessed: 2014-04-19]

[9] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," *Radar and Signal Processing, IEE Proceedings F*, vol. 140, no. 2, pp. 107–113, Apr. 1993.

[10] A. Doucet and A. M. Johansen, "A tutorial on particle filtering and smoothing: fifteen years later," in *The Oxford Handbook of Nonlinear Filtering*, 2011, pp. 656–704.

[11] J. S. Liu and R. Chen, "Sequential Monte Carlo Methods for Dynamic Systems," *Journal of the American Statistical Association*, vol. 93, pp. 1032–1044, 1998.

[12] C. M. Carvalho, M. S. Johannes, H. F. Lopes, and N. G. Polson, "Particle learning and smoothing," *Statistical Science*, vol. 25, no. 1, p. 88âĂŞ106, 2010.

[13] A. P. Sales, C. Challis, R. Prenger, and D. Merl, "Semi-supervised classification of texts using particle learning for probabilistic automata," in *Bayesian Theory and Applications*, P. Damien, P. Dellaportas, N. G. Polson, and D. A. Stephens, Eds. Oxford University Press, Jan. 2013.

[14] Transactional topologies in Storm. https://github.com/nathanmarz/storm/wiki/Transactional-topologies [Accessed: 2014-04-17]

[15] P. T. Goetz. Storm signals. https://github.com/ptgoetz/storm-signals [Accessed: 2014-04-18]

[16] Storm concepts. https://github.com/nathanmarz/storm/wiki/Concepts [Accessed: 2014-04-19]